Классы часто имеют тенденцию разрастаться по мере написания программы. Например, класс PointND на предыдущих занятиях приобретал такой вид:

```language-cpp
class PointND {
    unsigned total{ 0 };
    int* coords{ nullptr };
public:
    PointND() : total(0), coords(nullptr)
    { }
    PointND(unsigned sz) : total(sz)
    {
        coords = new int[total] {0};
    }
    PointND(int* cr, unsigned len) : PointND(len)
    {
        set_coords(cr, len);
    }
    PointND(const PointND& other) : PointND(other.coords, other.total)
    { }

    const PointND& operator=(const PointND& other)
    {
        delete[] coords;
        total = other.total;
        coords = new int[total];
        set_coords(other.coords, total);

        return *this;
    }

    unsigned get_total() { return total; }
    const int* get_coords() { return coords; }
    void set_coords(int* cr, unsigned len)
    {
        for (unsigned i = 0; i < total; ++i)
            coords[i] = (i < len) ? cr[i] : 0;
    }

    ~PointND()
    {
        delete[] coords;
    }
};
```

И это еще относительно небольшой класс. Если он и дальше продолжит увеличиваться, то программисту станет сложно в нем ориентироваться: искать нужные методы, вносить правки и т.п. Но как можно упростить это описание? Если мы вспомним с вами о прототипах функций, которые рассматривали в базовом курсе языка Си, то аналогичный подход можно было бы применить и при объявлении класса. Давайте посмотрим, как это делается.

Вначале мы опишем класс, который будет содержать объявления переменных, прототипы методов и методы с очень короткими реализациями:

```language-cpp
class PointND {
    unsigned total{ 0 };
    int* coords{ nullptr };
public:
    PointND() : total(0), coords(nullptr)
        { }
    PointND(unsigned sz) : total(sz)
        { coords = new int[total] {0}; }
    PointND(int* cr, unsigned len) : PointND(len)
        { set_coords(cr, len); }
    PointND(const PointND& other) : PointND(other.coords, other.total)
        { }
    const PointND& operator=(const PointND& other);

    ~PointND()
        { delete[] coords; }

    unsigned get_total() { return total; }
    const int* get_coords() { return coords; }
    void set_coords(int* cr, unsigned len);
};
```

Смотрите, здесь методы с большим количеством операторов представлены прототипами, остальные остались как есть непосредственно в классе. В результате текст программы стал гораздо понятнее и весь список методов буквально перед глазами программиста. Ориентироваться в таком классе куда проще.

Но где и как нам определить тела (реализации) для прототипов методов? Очевидно, это делается вне класса. **Как мы с вами уже говорили, класс формирует свою область видимости и для доступа к ней нужно прописать имя класса и воспользоваться символом четверототия (раскрытия области видимости)**:

```
[тип данных] <имя класса>::<элемент класса>
```

Первый прототип у нас – это оператор присваивания. За пределами класса к его описанию (прототипу) можно обратиться следующим образом:

```language-cpp
const PointND& PointND::operator=(const PointND& other)
{
        delete[] coords;
        total = other.total;
        coords = new int[total];
        set_coords(other.coords, total);

        return *this;
}
```

Обратите внимание, что **сначала указывается возвращаемый тип, затем, обращение к прототипу оператора в области видимости класса PointND и только потом – тело метода**. Таким образом, мы вынесли за пределы класса реализацию метода оператора присваивания.

По аналогии и со вторым прототипом set_coords:

```language-cpp
void PointND::set_coords(int* cr, unsigned len)
{
        for (unsigned i = 0; i < total; ++i)
            coords[i] = (i < len) ? cr[i] : 0;
}
```

В результате, мы получили точно такой же класс PointND, но с разнесенным его общим описанием и реализацией некоторых методов за пределами класса. Работать он будет абсолютно так же, как и прежде.

Обратите внимание, что формировать такое раздельное описание можно не только публичных методов, но вообще любых, например, приватных. Если метод set_coords в классе PointND поместить в секцию private:

```language-cpp
class PointND {
...
private:
    void set_coords(int* cr, unsigned len);
};
```

То никаких проблем с определением тела этого прототипа за пределами класса не будет, так как компилятор «понимает», что здесь прописывается

реализация метода, а не его вызов. А вот просто обратиться к приватному атрибуту set_coords уже не получится:

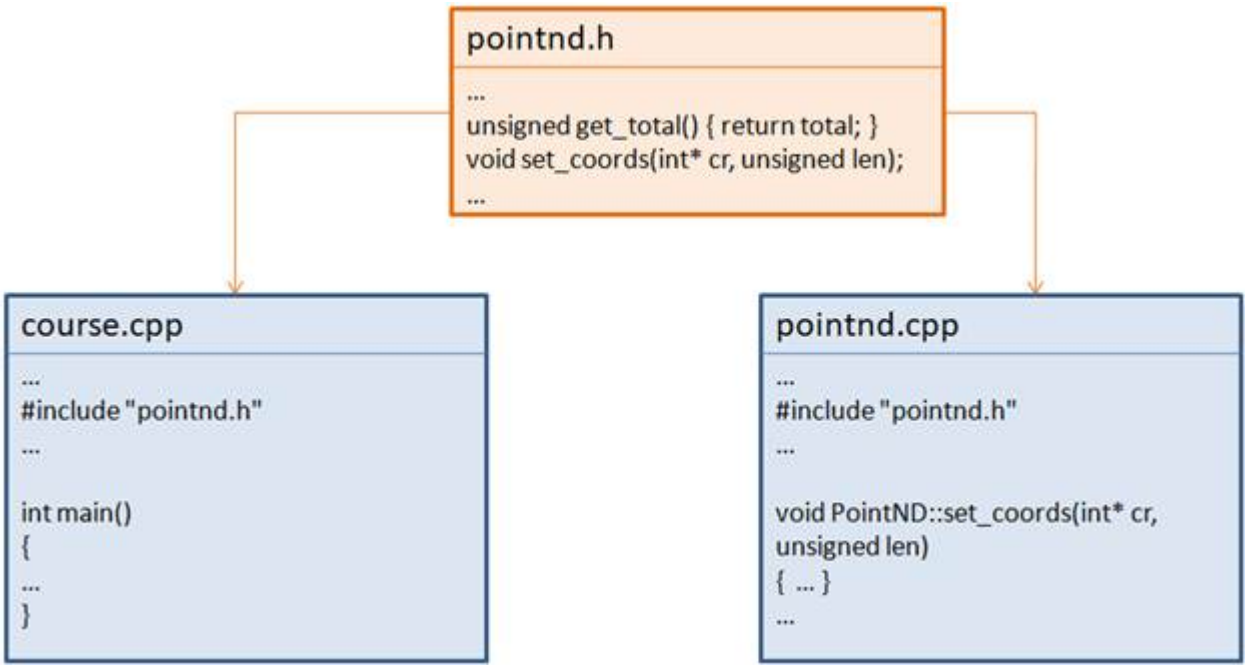
```language-cpp
int main()
{
    PointND pt(5);
    PointND::set_coords;   // ошибка
    PointND::get_coords;   // ok

    return 0;
}
```

Тогда, как с публичным атрибутом get_coords таких проблем не возникает. То есть, нужно различать описание метода и его вызов. Для компилятора – это принципиально разные вещи.

Многомодульные программы

Конечно, в реальных проектах объявления классов, как правило, делается в отдельных заголовочных файлах, а затем по мере необходимости, подключаются к файлам реализации.



В нашем примере было бы логично вынести объявление класса PointND в отдельный файл с таким же именем pointnd.h, а затем, подключить его к двум cpp-файлам:

- course.cpp – основная логика работы программы;
- pointnd.cpp – файл с реализациями прототипов методов.

Содержимое файла **pointnd**.***h** можно определить следующим образом:

```language-cpp
#ifndef _POINTND_H_
#define _POINTND_H_

class PointND {
    unsigned total{ 0 };
    int* coords{ nullptr };
public:
    PointND() : total(0), coords(nullptr)
        { }
    PointND(unsigned sz) : total(sz)
        { coords = new int[total] {0}; }
    PointND(int* cr, unsigned len) : PointND(len)
        { set_coords(cr, len); }
    PointND(const PointND& other) : PointND(other.coords, other.total)
        { }
    const PointND& operator=(const PointND& other);

    ~PointND()
        { delete[] coords; }

    unsigned get_total() { return total; }
    const int* get_coords() { return coords; }
    void set_coords(int* cr, unsigned len);
};

#endif
```

Обратите внимание на директивы условной компиляции. Они часто применяются в заголовочных файлах для защиты от повторного включения заголовка к одному и тому же cpp-файлу. Подробно мы с вами об этом говорили в базовом курсе языка Си.

Файл **course**.***cpp** определим как:

```language-cpp
#include <iostream>
#include "pointnd.h"

int main()
{
    int c[] = {1, 2, 3};
    PointND pt(c, 3);
```

```
    return 0;
}
```

А файл **pointnd**.***cpp** будет содержать реализации:

```cpp
#include "pointnd.h"

const PointND& PointND::operator=(const PointND& other)
{
        delete[] coords;
        total = other.total;
        coords = new int[total];
        set_coords(other.coords, total);

        return *this;
}
```

```cpp
void PointND::set_coords(int* cr, unsigned len)
{
        for (unsigned i = 0; i < total; ++i)
            coords[i] = (i < len) ? cr[i] : 0;
}
```

Все, у нас с вами получилась программа, состоящая из двух модулей и одного заголовочного файла pointnd.h.

Но, давайте, детальнее посмотрим, как будет подключаться класс в каждый из этих модулей. В классе PointND есть определение методов вместе с их реализациями. **Так как тела методов достаточно просты, то часто компилятор превращает их в inline-методы, то есть, они не вызываются подобно функциям, а их реализации буквально вставляются в места их вызова**. Но, если какой-либо метод компилятор воспринимает на уровне обычной функции и реализация этого метода прописана в классе, то тело такого метода будет продублировано в каждом модуле программы, где есть подключение заголовка с этим классом. Конечно, это приведет к увеличению размера самой программы, хотя других последствий быть не должно. Программа в целом будет работать корректно. Поэтому большие реализации все же лучше выносить за пределы класса и объявлять их в отдельном модуле. В нашем примере – это модуль pointnd.cpp.